Re: review materials (Re: ADDI-DATA Driver Review)

Till Straumann <strauman@slac.stanford.edu>

Wed 3/9/2016 1:12 PM

To: Ford, Christopher <caf@slac.stanford.edu>; D'Ewart, J. Mitch <mdewart@slac.stanford.edu>; Kim, Kukhee <khkim@slac.stanford.edu>;

Here's a (simplified) example from their apci1710ctr.c (kernel module)

```
static bool ringbufPush(counter_channel_t *pchan, uint32_t counter)
{
  int tmpRead, tmpWrite, tmpIndex, frameCountTmp;
  bool  rv;

  tmpRead  = atomic_read(pchan->readIndex);
  tmpWrite = atomic_read(pchan->writeIndex);

  tmpIndex = (tmpWrite + 1) % _ringSize;

  if (tmpIndex == tmpRead) {

    /* buffer full! update the overflow counter */
    overflowCountIncrement(pchan);
    rv = false;

  } else {

    /* update the element */
    pchan->ringBuf[tmpIndex].counter = counter;

    /* update write index */
    atomic_set( &pchan->writeIndex, tmpIndex);
    rv = true;
  }
  return rv;
}


static bool ringbufPop(counter_channel_t *pchan, uint32_t *counter, uint16_t *frameCount)
{
  int   tmpIndex, tmpRead, tmpWrite;
  bool  rv = false;

  tmpRead  = atomic_read(pchan->readIndex);
  tmpWrite = atomic_read(pchan->writeIndex);

  if (tmpWrite != tmpRead) {
    tmpIndex    = (tmpRead + 1) % _ringSize;
    *counter    = pchan->ringBuf[tmpIndex].counter;
```

```
    *frameCount = pchan->ringBuf[tmpIndex].frameCount;

    /* update read index */
    atomic_set( &pchan->readIndex, tmpIndex );

    rv = true;
  }
  return rv;
}
```

There are multiple problems here:

neither the 'push' nor the 'pop' routines are thread-safe (as discussed already). Because the indices are not really manipulated in an atomic fashion (that would require an atomic_inc_modulo_size() operation) multiple threads can mess up the indices and not only the data.

This particular module allows 'pop' from user-mode and 'push' from user-mode *and* the interrupt handler. Certainly not 100% safe

If you want to atomically manipulate the read pointer then:

```
   item * pop(channel *pchan)
   {
   item *rval;

   unsigned oldIdx = atomic_read( &pchan->readIndex );
   unsigned newIdx, expected;
   do {

     if ( oldIdx == atomic_read(pchan->writeIndex )
        return NULL; // buffer empty

     if ( (newIdx = oldIdx + 1) == ringSize )
        newIdx = 0; // wrap around

     rval = pchan->buffer[newIdx];

     expected = oldIdx;

     oldIdx   = atomic_cmpxch( &pchan->readIndex, oldIdx, newIdx );

   } while ( oldIdx != expected );

   return rval;
   }
```

Looks better - does it? Well this implementation still suffers from the so-called ABA problem: what happens if

1) thread A reads 'oldIdx and determines it is != writeIdx (buffer not empty)
2) other threads preempt A, push and pop threads until 'oldIdx' again holds the same value A had read during 1).

```
    Assume the buffer is empty at this point (readIdx == writeIdx).
3)  thread A resumes, the atomic_cmpxch() succeeds replacing 'oldIdx' with 'newIdx'
    thereby overflowing the buffer (readIdx now past writeIdx).

This can be mitigated if you e.g., require the buffer size to be a power of two.
Then:



   item * pop(channel *pchan)
   {
   item *rval;

   unsigned oldIdx = atomic_read( &pchan->readIndex );
   unsigned newIdx, expected;
   do {

     if ( oldIdx == atomic_read(pchan->writeIndex )
        return NULL; // buffer empty

     rval = pchan->buffer[newIdx & (RING_SIZE-1)];

     expected = oldIdx;

     oldIdx   = atomic_cmpxch( &pchan->readIndex, oldIdx, newIdx );

   } while ( oldIdx != expected );

   return rval;
   }

Now the ABA problem would require thread A
to preempted until the full range of readIndex (e.g. 2^32) is exhausted
(since the modulo 'ring_size' is only taken when accessing the buffer).

Obviously a 'push' for multiple writers is more complex since you cannot
re-try the store-to-buffer operation.

There are yet more problems: the algorithm lacks proper memory barriers.

If e.g. (assume single-writer only; no overflow check)

    push:
      newIdx = atomic_read( &pchan->writeIdx ) + 1;
      buf[newIdx & (RING_SIZE - 1 ) ] = newItem;
          /*** write-barrier HERE must ensure buf[] is written BEFORE 'newIdx' ***/
      atomic_set( newIdx, &pchan->writeIdx );


and you have two threads executing on different CPUs; one pushing the
other popping, then, without memory barriers, there is no guarantee
that the 'popping' CPU observes the updated value in the 'buffer' array.
It could instead obtain the 'old' value. (A read barrier must be
inserted between reading the writeIdx and the buffer contents).
```

```
I cooked this all up in half an hour and am far from certain that
the presented examples are correct. Just wanted to illustrate that
lock-free design is far, far, from trivial. Unless performance is
really critical and seriously impacted by locking algorithms
then I'd stay away from lock-free solutions -- unless you can pull
them out of a good-quality library (in user-mode, e.g., boost).
```

- T.

On 03/08/2016 04:32 PM, Ford, Christopher wrote:

> See you tomorrow at the ADDI-DATA driver review.
>
> Thanks,
> -caf